Spring 4-1996

Design of a Microelectronic System for Multiple Technologies Xilinx and Actel

Brian Christopher Mann
University of Tennessee - Knoxville

Design of a Microelectronic System for Multiple Technologies (Xilinx and Actel)

SENIOR PROJECT REPORT

University Honors Program

April 14, 1996

Brian C. Mann

Electrical & Computer Engineering

University of Tennessee

Knoxville, TN 37996-2100

Contents

List of Figures

ABSTRACT

This project involved the design, simulation, realization and demonstration of a tic-tac-toe game using Xilinx and Actel Field Programmable Gate Arrays (FPGAs). The behvavior of the game is described in a hardware description language (VHDL), which is synthesized into a structural description and combined with other components. After simulation verified proper operation, the design could be physically placed and routed using technology-dependent tools. Using this synthesis-based approach, most of the design is captured only once and then can be implemented in multiple technologies.

1 Introduction

This project involved the design of a microelectronic system to meet the requirements of a specific application. The system consisted of input/output devices coupled with digital logic circuitry implemented in a single application-specific integrated circuit (ASIC). The design had to be specified, as shown in Figure 1, at every level of abstraction including the system, behavioral, structural, and physical levels. The design process involved mapping the requirements of the application into lower level specifications that detail not only the internal functions but also the interactions among the components and the external world.

For this project, a hardware description language, VHDL, was used to specify portions of the design. This representation was then synthesized and mapped into a chosen technology. The resulting structural (logic) description was then combined with other components which had been specified using a schematic. The design was then simulated for functionality. Once verified, the structural representation could be translated into a physical representation that was placed and routed automatically. The design could then be resimulated to verify proper operation and timing.

Using this synthesis-based approach, most of the design was captured only once and then an attempt was made to implement in multiple technologies. In this case, the technologies included field-programmable gate arrays (FPGAs) supplied by Xilinx and Actel. Standard-cells could also be fabricated as a single integrated circuit (IC) via MOSIS if desired. Thus, the various implementations could be compared based on area, delay, cost, turnaround time, etc.

The design was initially implemented in a Xilinx FPGA since it is reprogrammable and allows changes to be made, if required. This part was placed on a printed circuit board (Xilinx Demonstration Board) along with the associated input/output devices to demonstrate proper operation. Implementation using the one-time programmable Actel FPGA could then be performed to provide a definitive constrast to the Xilinx

FPGA. These two families of FPGAs differ significantly in their architecture, logic resources, interconnect resources and physical routing software. Consequently, a design must be physically implemented in both technologies to make a fair comparison.

The FPGA implementations could serve as rapid prototypes for a standard-cell integrated circuit fabricated via MOSIS. Only after verification of the FPGA parts should the design be submitted for fabrication since the turnaround time is generally eight weeks. MOSIS merges projects from several organizations onto a single set of masks and wafers, thereby sharing the costs and making four copies of each project cost only $500. The National Science Foundation sponsors the fabrication of educational projects so that there is no cost to student designers.

This report is organized along the lines of the design flow shown in Figure 1. First, the system requirements and then the detailed specifications are given. Next, the behaviorial and structural representations are presented. The physical layouts are then implemented for each technology described.

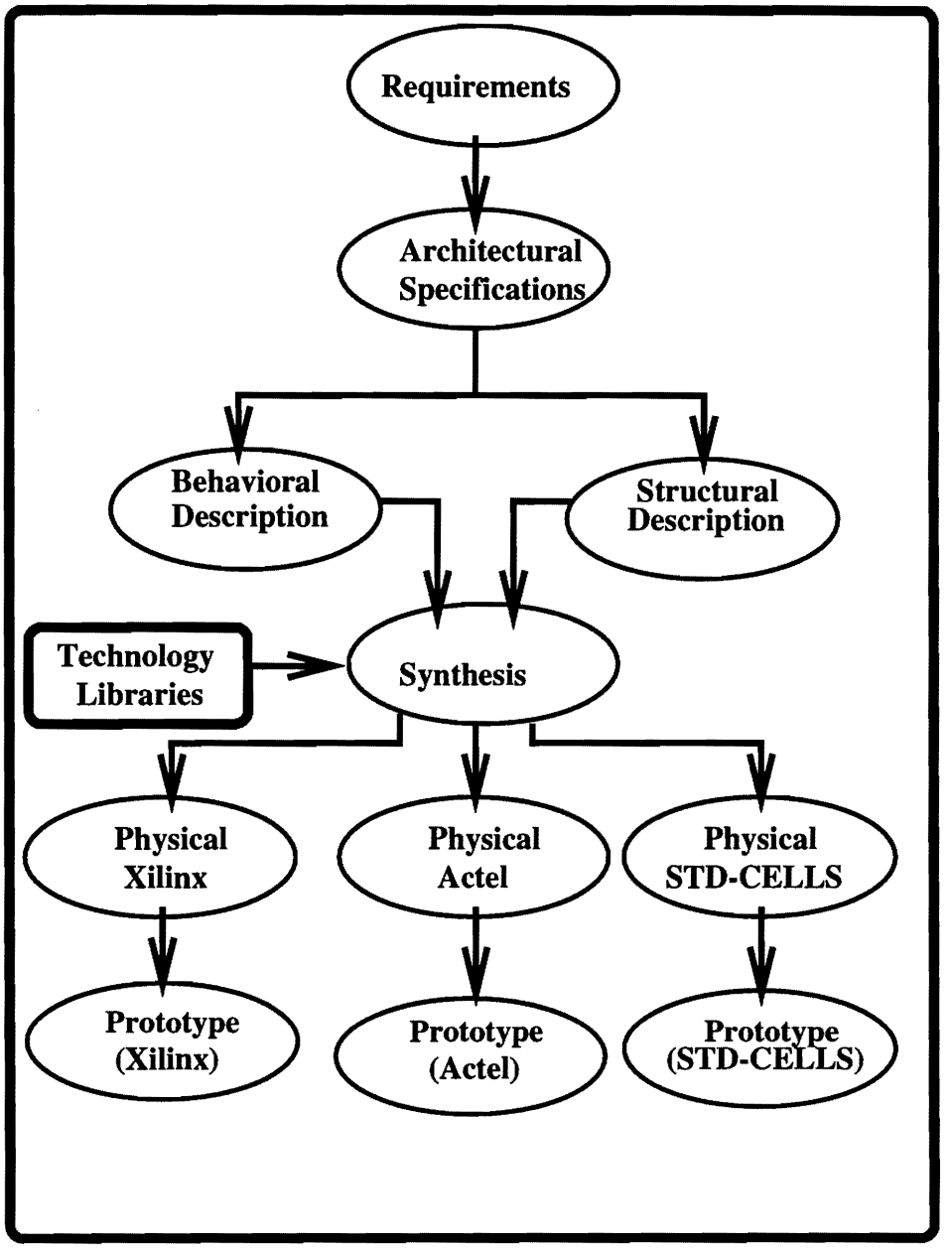


Figure 1: *Design of a Microelectronic System for Multiple Technologies.*

2 System Requirements

The microelectronic system designed for this project consisted of a tic-tac-toe game. The game is designed for one player to play against a second player in the form of a Xilinx 4005 FPGA. The tic-tac-toe grid is represented by a group of LEDs. There are nine sets of two LEDs, each representing either the X or the O player. The user will begin play and chooses his/her space on the grid by depressing a button. Once depressed, a colored LED (green or yellow) will light signifying the location selected. Once the user(player X) has moved, it will then be O's turn. The ASIC, in turn, will perform a logic algorithm and determine a 'smart' location with which to counter. Play will then be returned to X. If player X accidenatally chooses an occupied position on the board, an LED will light signifying an error. The player may then choose another position. Play will continue in this manner until either a win or a draw has resulted. If a player has successfully won the game, an appropriate LED will light signifying a win. Likewise a separate LED will light signifying a draw when the draw condition occurs. An external reset button may be depressed at any time to begin a new game. Figure 2shows a system-level block diagram of the project.

The requirements described above are known as the MAIN portion of the project. The requirements for built-in self-test are known as the BIST portion. The BIST portion of the circuit lights up each of the LED's after recognizing user input. In this manner the I/O operation can be verified. The BIST portion will be loaded to the ASIC separately from the main portion.

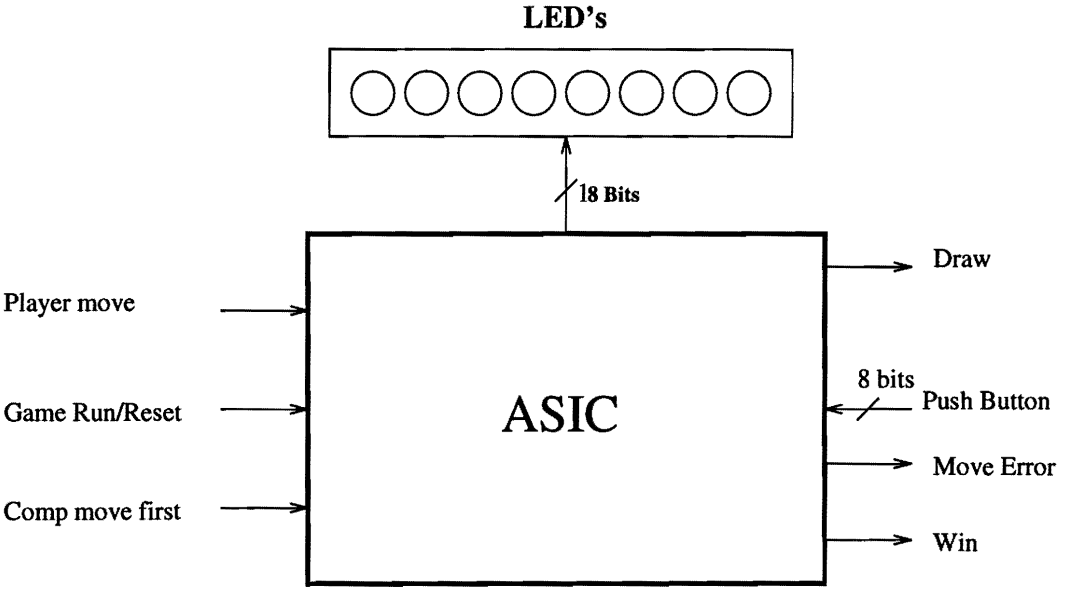


Figure 2: *System-Level Block Diagram.*

3 ASIC Specifications

The design consists of several functional blocks, each of which performs a particular task. Figure 3 shows the I/O and internal components of the ASIC. The Player Module represents user input from the nine position select switches and provides logic for an alternating player sequence. The Conrol Module is a state machine which controls the sequence of the game. The Control Module reads flags from other condition-detect devices to determine the next state of the device. Resources such as the Winner/Draw detect the conditions that the control module requires for proper state succesion. The Board Control Module is responsible for the representation of the tic-tac-toe grid in the form of LED's. An external reset is used to begin a new game.

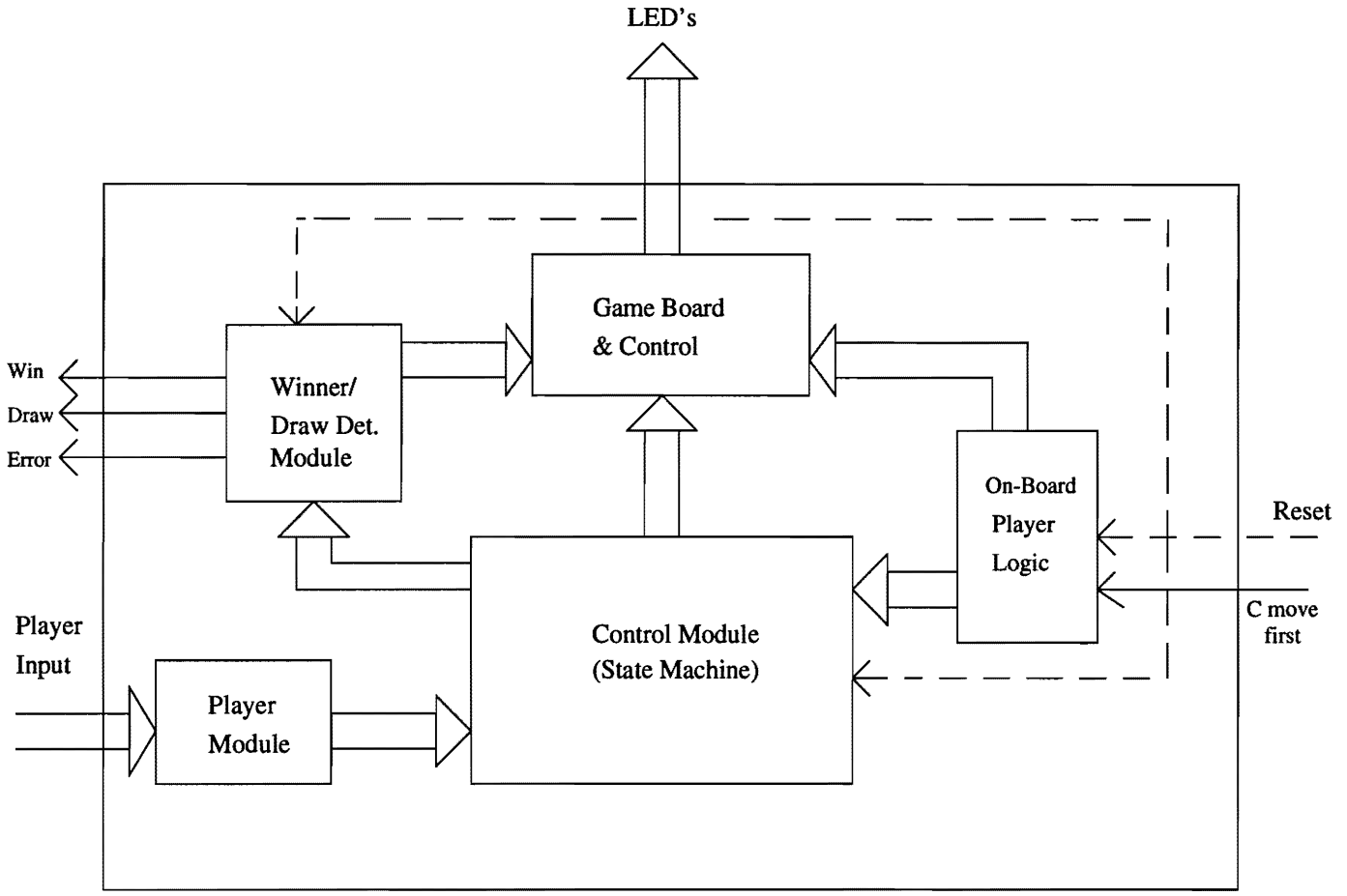


Figure 3: *ASIC I/O and Internal Components.*

4 Xilinx Implementation

Figure 4 shows the arrangement of the function blocks along with the Xilinx-specific input and output pads. Figure 5 shows the simulation in VIEWSIM of the Board control module prior to physical layout. In this simulation, the activity of the playing board can be seen. In this figure, C and P represent the computer and player outputs respectively. Figure 6 shows the simulation of the control module. This figure demonstrates the response to test stimulus of this module. VIEWSIM was used to perform a simulation of all of the modules designed in this project. The sample simulation included, along with others, verified the proper operation of the modules used in the project. All of the modules were then connected together in the VIEWDRAW environment resulting in a structural description of the design.

The structural description (net-list) was then converted into the Xilinx XNF format. This file was then processed by the Xilinx XACT software to assign specific logic functions to Combinational Logic Blocks (CLBs) in the Xilinx 4005 part. At this point in the project a problem arose. XMAKE could not produce a layout for this design. The problem seemed to lie in the wiring in VIEWDRAW. Knowing this, a VHDL file was written to perform the interconnection between modules. This not only corrected the problem in XMAKE, but also simplified the process of changing technologies in the future.

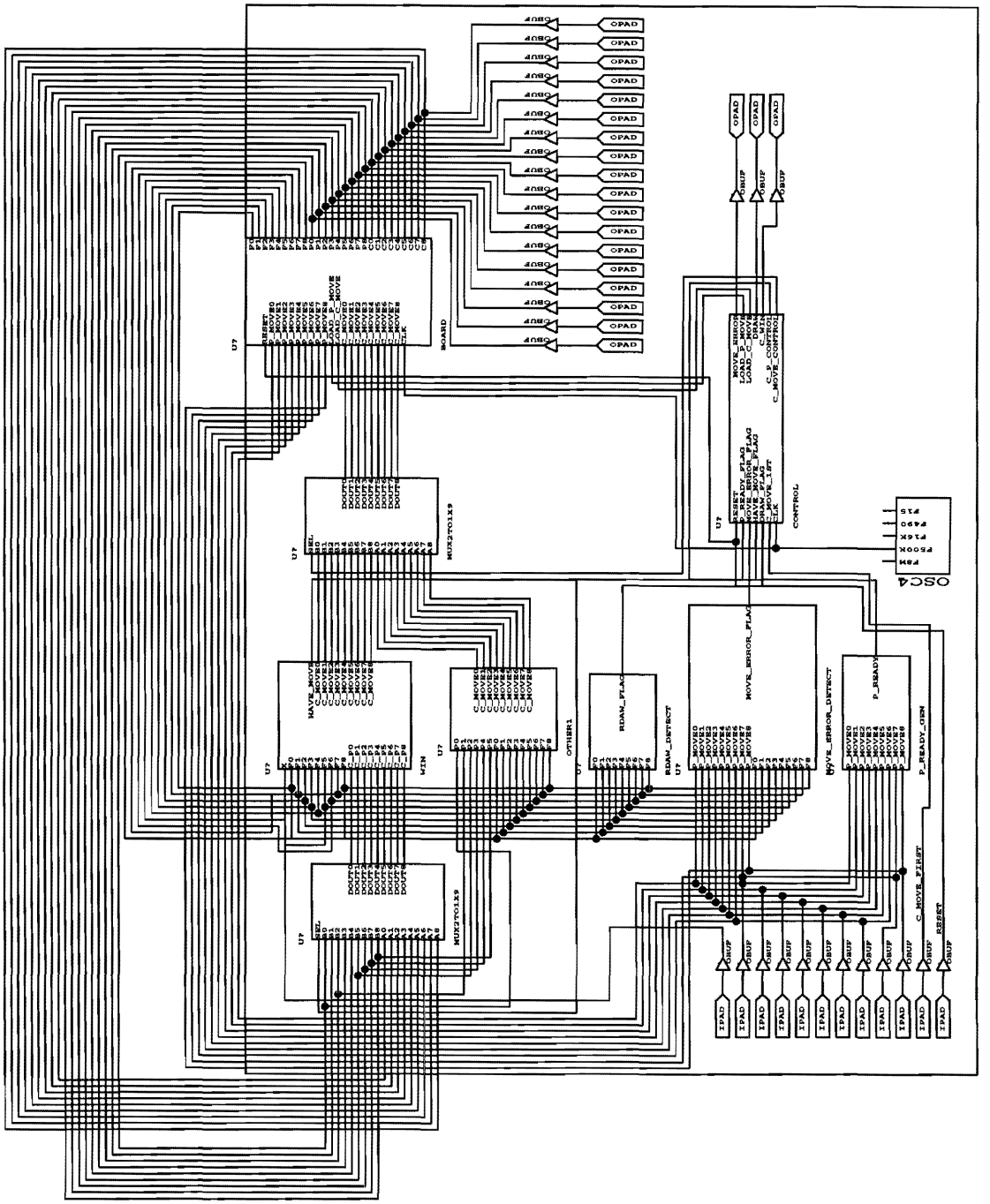


Figure 4: *I/O and Internal Components for Xilinx 4005.*

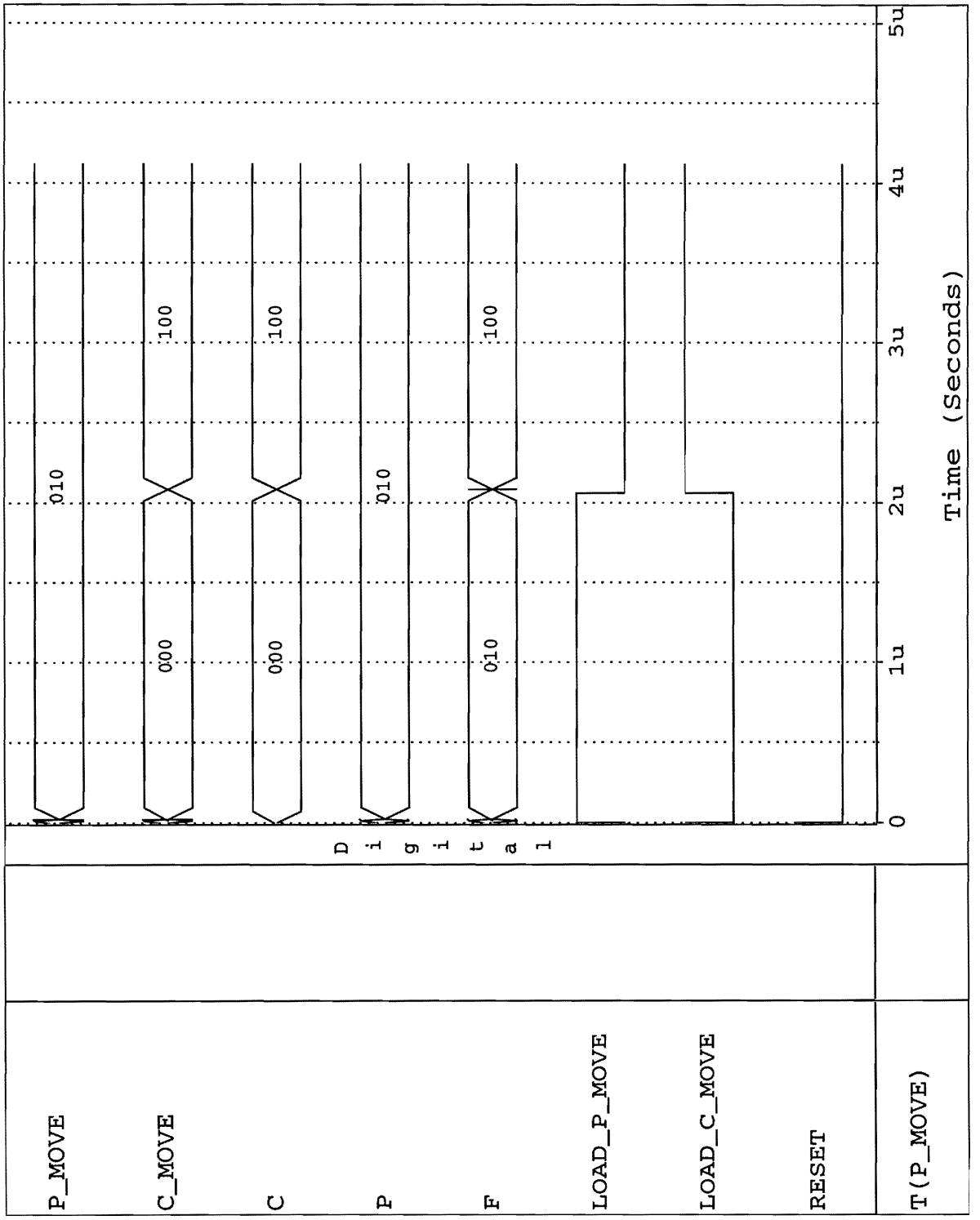


Figure 5: *Pre-layout Simulation for Xilinx using VIEWSIM.*

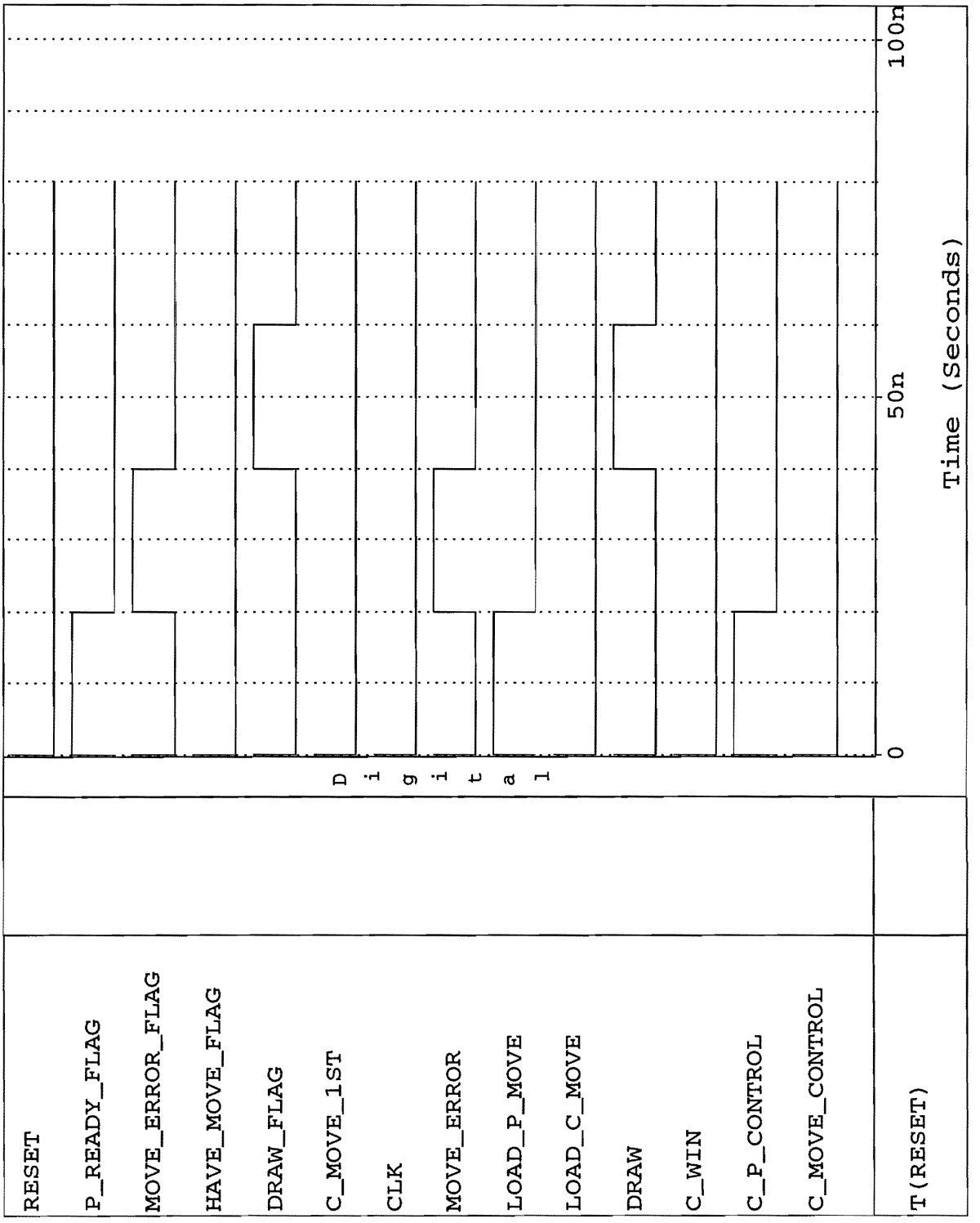


Figure 6: *Pre-layout Simulation for Xilinx using VIEWSIM.*

5 Actel Implementation

Since the function blocks were developed previously using technology-independent VHDL, they could have been resynthesized into a technology-dependent structural description using the Actel library. The resulting logic could also be simulated using VIEWSIM prior to layout as was done for the Xilinx technology.

The Actel ALS software could be used to perform the physical placement and routing of the design to the Actel technology. A post-layout simulation could then be performed for analysis purposes.In general, this post-layout simulation could be used to study the critical timing paths and to help the designer optimize the design to meet speed requirements. Due to time constraints, the Actel solution was not implemented in this case.

6 Layout

With the Xilinx solution fully designed and simulated, a final layout could be performed. The goal of the layout process is to physically place and route every CLB in a manner to optimize the design. When optimizing, two aspects are considered: speed and area. These two constraints are inversly proportional to each other. An increase in speed requires greater area in the layout and vice-versa. In the case of FPGA's and standard cell designs, greater area costs more money. The goal is to find a convenient trade-off between area and speed for the particualr design that you are working on.

VIEWlogic will allow the user to place and route the design multiple times using iterative techniques. All solutions to the iterations are kept in a log file. The designer can allow the software to optimize for a certain amount of time, review the log file, and select the best solution from the list for placement and routing of the CLB's.

In the design of the tic-tac-toe game, speed is of little importance. The reaction time of the user will be magnitudes larger than the decision time of the logic within. For this reason the design was mapped for optimum or smallest area. The resulting schematic digram after place and route is found in Figure 7. This result implements eleven rows of interconnected CLB's, approximately 2200 logic gates.

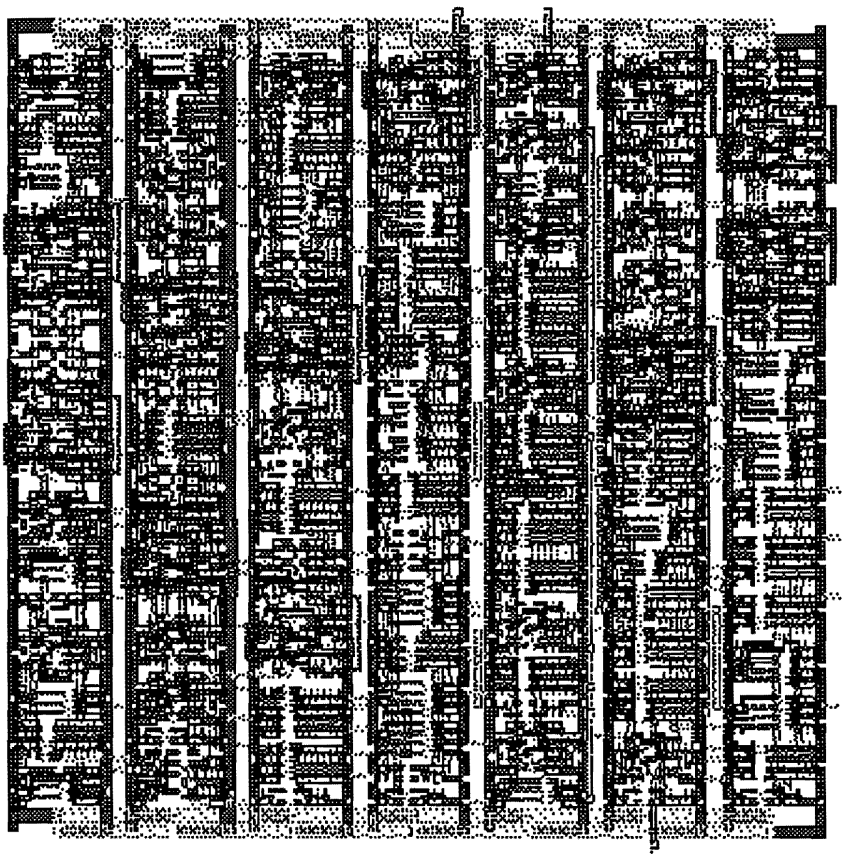


Figure 7: *XMAKE Place and Route Schematic*

7 Results

WIth a final layout completed and optimized, the design was prepared for download into a Xilinx 4005 Field Programmable Gate Array. LED's representing the player grid were soldered onyo the FPGA. Switches were also provided for player interface. The specific pins used for interface are designated by the designer and are shown previously in Figure 4.

One goal of this project was to implement the design into two technologies, Xilinx and Actel. Since these two technologies differ greatly in their logic and interconnect farmework, comparisons were to be made on the resulting timing considerations and of the layout area consumed by the design on the two FPGA's. In this design, the best comparison would be on area consumed. Due to time constraints, only the Xilinx solution was implemented.

The design was downloaded into the Xilinx 4005 from a workstation via the XACT software package. Transmission proceeded with no problem. However, a problem was encountered upon execution of the game. The player was able to begin the game and select a starting position on the board. The FPGA would then generate a move to block. At this point the game "locked up" and allowed no further input. A timing problem existed in the player control model of the game. The timing problem arose due to the timing delays inherrent in the CLB blocks. These timing delays were ignored by the designer as time did not seem critical in the execution of a tic tac toe game. Speed is not important in this case. But synchronization of logic between modules is very important. The timing delays inherrent in the CLB's caused this synchronization to be slightly off, resulting in malfunction of the design.

8 Summary and Conclusions

The tic-tac-toe game proved to be quite a challenge. The design approach and considerations were changed many times as more and more was learned about VHDL design. The basic design approach used is summarized below.

VHDL was used to successfully generate nine working modules for the game. The VIEWGEN tool then generated a schematic and symbol for each module. VIEWDRAW was the used to connect these modules as designed. VSM successfully created the net-list for the design. Xmake was successful in performing the place and route for the design. The design was implemented in the Xilinx 4005 technology but was unsuccessful in operation. This failure was the result of improper timing considerations of the designer when constructing the VHDL for the design. Timing considerations have proven to be the most difficult part of using VHDL. It is more difficult to specify synchronization and timing in a hardware language than by using standard off the shelf parts. The timing delays of off the shelf parts are known with great accuracy, where as the timing considerations of an FPGA are specific to the exact design being implemented. With more amd more experience with VHDL and FPGA design, these sychronization problems can be avoided.

It is regretful to report that the project was not a complete success. If the Xilinx implementation had been a success, the design could then be retargeted to the Actel technology for comparisons on speed, area, and performance. The amount of knowledge gained from the design, though, has been enormous. VHDL is a very powerful design tool. Complex state machines can be built using only a few lines of VHDL code. The VIEWLogic toolset and VHDL make implementing a design in multiple FPGA technologies a simple task. The only limit found was time.

References

[1] Devine, Quinn, "Synthesis Design Flow Tutorial", Knoxville, TN., Microelectronics Research Group,The University of Tennessee, Knoxville, http//microsys6.engr.utk.edu/ devine/tutorial/tut.html (1996).

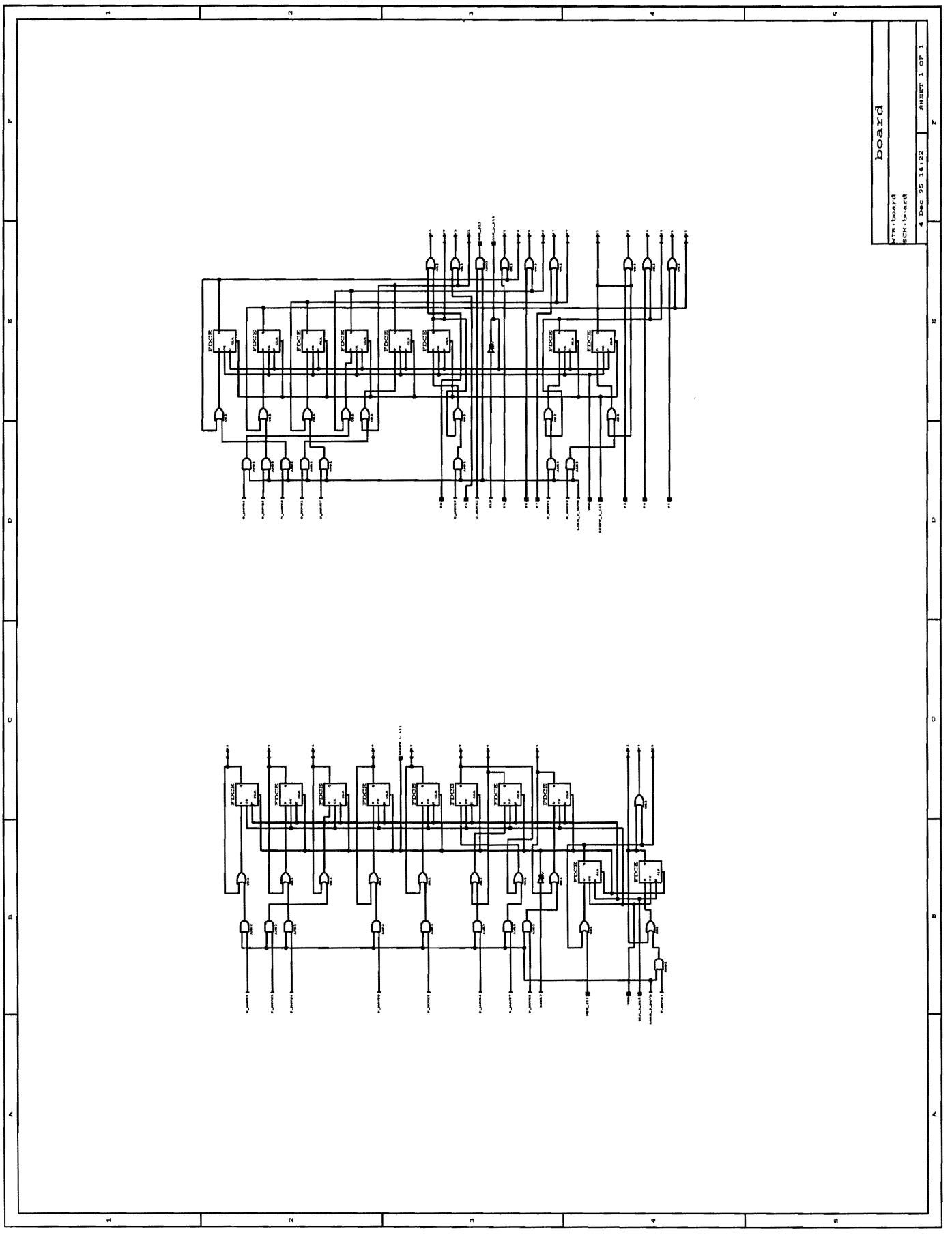
[2] Bouldin, D., "VLSI Systems Design", Chapter 13 in Computer Engineering Handbook Chen, C. H. (ed.), New York: McGraw-Hill, pp. 13.1-13.11 (1992).

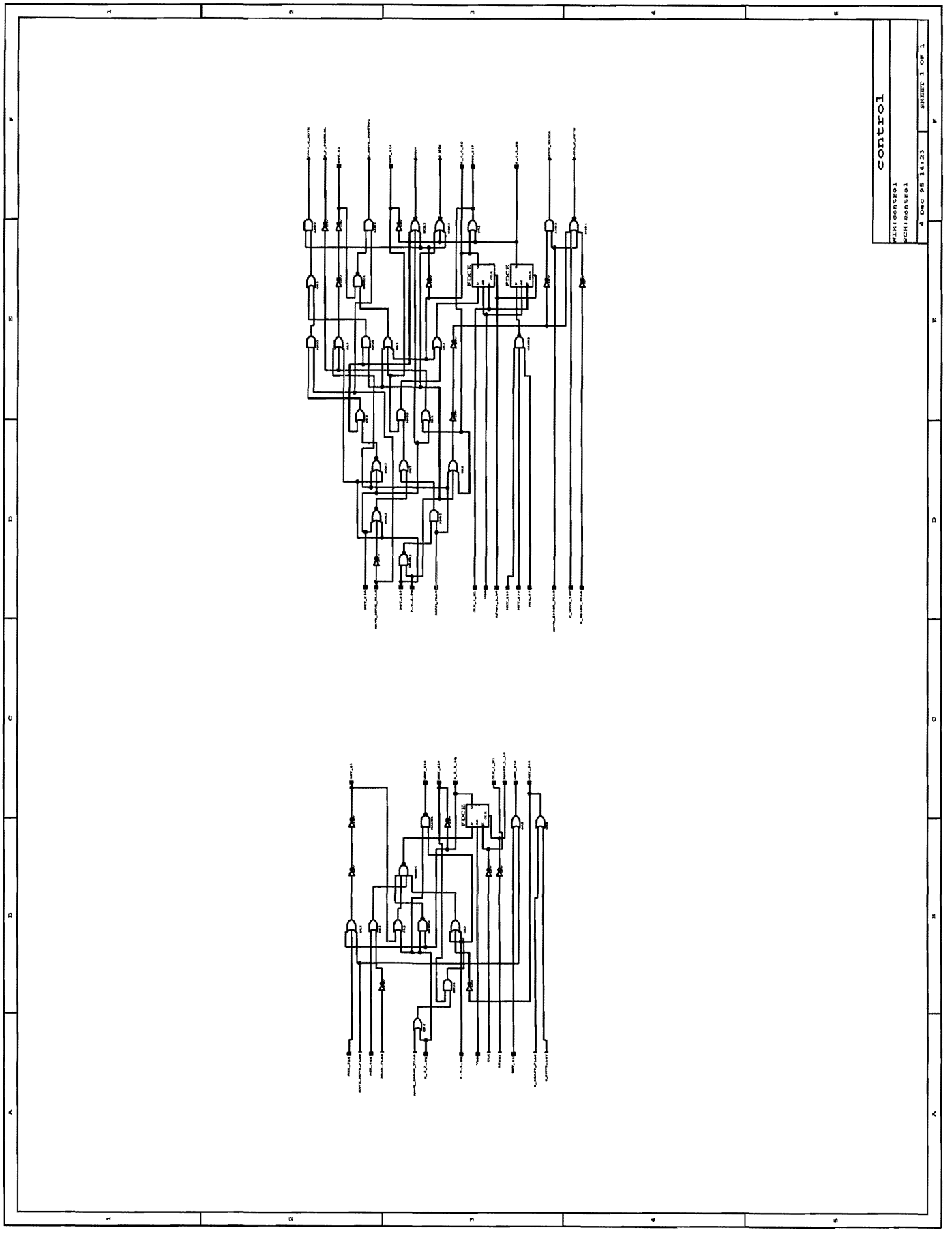
[3] Cooley, J., "True Lies in FPGA Synthesis Benchmarking", EDN, pp. 38-49 (October 27, 1994).

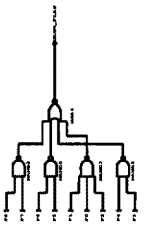
[4] Bagri, A., "Automated Evaluation and FPGA Implementation of VHDL-Based Designs", M. S. Thesis, Knoxville,Tn.,Electrical and Computer Engineering, University of Tennessee (May 1994).

[5] Bouldin, D., "VLSI Designers' Interface Column", IEEE Circuits and Devices Magazine, vol. 9, no. 5, pp. 5-6, (September 1993).

9 APPENDIX A - Schamatic Diagrams

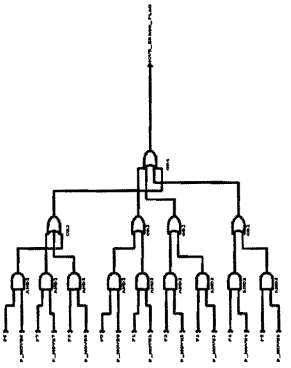

board
4 Dec 95 14:22
SHEET 1 OF 1


control
MIR:control
SCH:control
4 Dec 95 14:23
SHEET 1 OF 1



draw_detect		
WIR:draw_detect		
SCH:draw_detect		
4 Dec 95 14:25		SHEET 1 OF 1

move_error_detect		
HIR:move_error_detect BCH:move_error_detect		
4 Dec 95 20:12		SHEET 1 OF 1

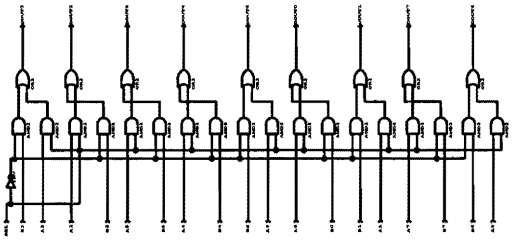


mux2to1x9

PIR:mux2to1x9
SCH:mux2to1x9

4 Dec 95 14:28

SHEET 1 OF 1



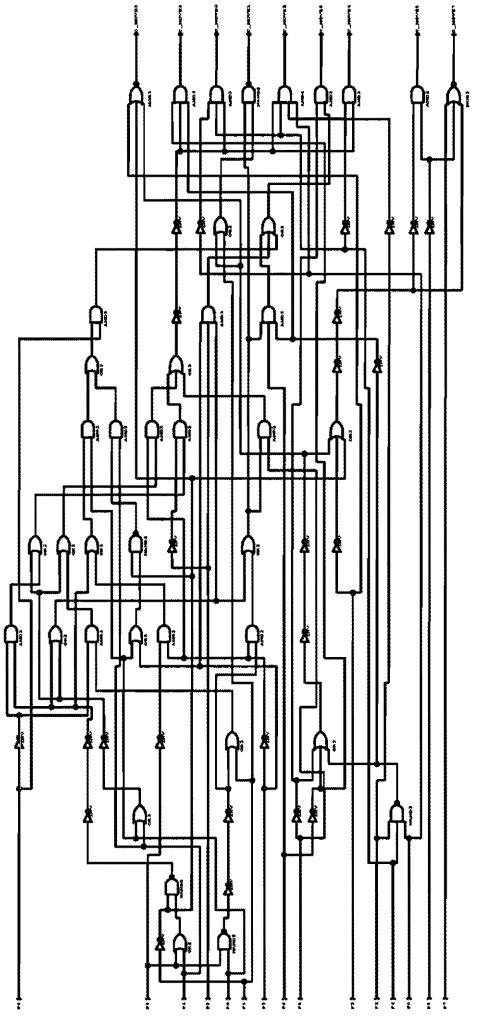
other

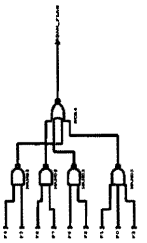
WIR:other

SCH:other

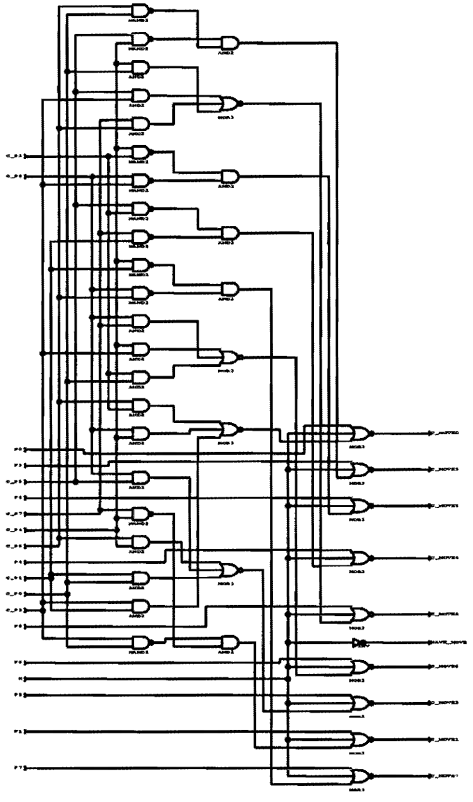
7 Dec 95 20:15

SHEET 1 OF 1





rdaw_detect		
WIR:rdaw_detect		
SCH:rdaw_detect		
4 Dec 95 14:41		SHEET 1 OF 1



win	
WIR:win	
SCH:win	
7 Dec 95 17:42	SHEET 1 OF 1

10 APPENDIX B - VHDL code

```
-- vhdl model for the board

-- include libraries for the flip-flops here
library synth,work;
use synth.stdsynth.all,work.techdef.all;
                  -- note that dffc is a single ff and dffc_v is a vectored ff

--*************************************************************************
-- IO Interface Declaration
--*************************************************************************

entity board is
port(
  signal    clk:         in  vlbit;
  signal    c_move:      in  vlbit_1d(8 downto 0);
  signal    load_c_move: in  vlbit;
  signal    load_p_move: in  vlbit;
  signal    p_move:      in  vlbit_1d(8 downto 0);
  signal    reset:       in  vlbit;
  signal    c:          inout  vlbit_1d(8 downto 0);
  signal    f:           out  vlbit_1d(8 downto 0);
  signal    p:          inout  vlbit_1d(8 downto 0));
end board;

--*************************************************************************
-- Architecture body
--*************************************************************************

architecture behavior of board is


-- declare internal signals here
signal c_temp : vlbit_1d(8 downto 0);
signal p_temp : vlbit_1d(8 downto 0);

signal reset_l : vlbit;
signal clk_l : vlbit;

begin

-- the following is done to match the dff_v ff to what we want to use
-- the dff_v is active high reset and rising edge triggered
-- we want active low reset and falling edge trigerred
-- this will insure proper mapping to the stdcell dfrf301
-- without ant extra gates to change the reset to active low etc...

-- this is to convert from active high reset to active low
reset_l <= not reset;

-- this is to convert from rising edge clock to falling edge
clk_l <= not clk;

-- these are the flip-flops in the synth library we are to use

-- this one is for the comp
dffc_v(c_temp,reset_l,clk_l,c);

-- this one is for the player
dffc_v(p_temp,reset_l,clk_l,p);

-- note we do not need one for the full


process (c, p, c_move, p_move, load_c_move, load_p_move, c_temp, p_temp)
-- include all inputs and inouts in the process sensitivity list
```

```
begin

      if (load_c_move = '1') then
         c_temp <= (c or c_move);
      else c_temp <= c;  -- note we need to pass the value if it doesnt change
                         -- if we dont then the output c_temp is latched
                         -- which is before the ff we want c after the ff
      end if;

      if (load_p_move = '1') then
         p_temp <= (p or p_move);
      else p_temp <= p;  -- note we need to pass the value if it doesnt change
                         -- if we dont then the output p_temp is latched
                         -- which is before the ff we want p after the ff

      end if;

      f <= c or p;

-- or it could be f <= c_temp or p_temp;
-- the first puts the 'or' after the ff and the last puts it before
-- this is an example of how vhdl does exactly what you tell it
-- so be smart in writing your code

end process;
end behavior;
```

```
-- vhdl model for the control

-- include libraries for the flip-flops here
library synth;
use synth.stdsynth.dffc_v;

--*************************************************************************
-- IO Interface Declaration
--*************************************************************************

entity control is
port
(
-- inputs
  signal  clk:              in  vlbit;
  signal  c_move_1st:       in  vlbit;
  signal  draw_flag:        in  vlbit;
  signal  have_move_flag:   in  vlbit;
  signal  move_error_flag:  in  vlbit;
  signal  p_ready_flag:     in  vlbit;
  signal  reset:            in  vlbit;

-- outputs
  signal  c_move_control:  out  vlbit;
  signal  c_p_control:     out  vlbit;
  signal  c_win:           out  vlbit;
  signal  draw:            out  vlbit;
  signal  load_c_move:     out  vlbit;
  signal  load_p_move:     out  vlbit;
  signal  move_error:      out  vlbit
);
end control;

--*************************************************************************
-- Architecture body
--*************************************************************************

architecture behavior of control is


-- declare internal signals here
signal n_state : vlbit_1d(2 downto 0);
signal p_state : vlbit_1d(2 downto 0);

signal reset_l : vlbit;
signal clk_l : vlbit;


-- state assignments are as follows
constant       p_move_state: vlbit_1d(2 downto 0) := B"000";
constant    c_win_chk_state: vlbit_1d(2 downto 0) := B"001";
constant  c_block_chk_state: vlbit_1d(2 downto 0) := B"010";
constant other_c_move_state: vlbit_1d(2 downto 0) := B"011";
constant        c_win_state: vlbit_1d(2 downto 0) := B"100";
constant         draw_state: vlbit_1d(2 downto 0) := B"101";


begin

-- this is to convert from active high reset to active low
reset_l <= not reset;

-- this is to convert from rising edge clock to falling edge
clk_l <= not clk;

-- these are the flip-flops in the synth library we are to use
```

```
-- this is for the state registers
dffc_v(n_state,reset_l,clk_l,p_state);


state_machine: process (have_move_flag, draw_flag, move_error_flag, p_ready_flag, c_move_1st,
begin

-- defaults here
-- very important to put the defaults here if not then once set they remain set

load_c_move <= '0';
load_p_move <= '0';
move_error <= '0';
c_win <= '0';
draw <= '0';
c_p_control <= '0';
c_move_control <= '0';
n_state <= p_state;

if (p_state = p_move_state) then
   if (draw_flag = '1') then
      n_state <= draw_state;
   elsif (move_error_flag = '1') then
      move_error <= '1';
      n_state <= p_state;
   elsif (c_move_1st = '1') then
      n_state <= c_win_chk_state;
   elsif (p_ready_flag = '1') then
      load_p_move <= '1';
      n_state <= c_win_chk_state;
   end if;

elsif (p_state = c_win_chk_state) then
-- choose c on c_p bus
      c_p_control <= '1';
      if (p_ready_flag = '0' and c_move_1st = '0') then
         if (draw_flag = '1') then
            n_state <= draw_state;
         elsif (have_move_flag = '1') then
-- choose the win/block move
            c_move_control <= '1';
            load_c_move <= '1';
            n_state <= c_win_state;
         else
            n_state <= c_block_chk_state;
         end if;
      end if;

elsif (p_state = c_block_chk_state) then
-- choose p on c_p bus
      c_p_control <= '0';
      if (have_move_flag = '1') then
-- choose win/block move
         c_move_control <= '1';
         load_c_move <= '1';
         n_state <= p_move_state;
      else
         n_state <= other_c_move_state;
      end if;

elsif (p_state = other_c_move_state) then
-- choose  other move
      c_move_control <= '0';
      load_c_move <='1';
      n_state <= p_move_state;
```

```
elsif (p_state = c_win_state) then
      c_win <= '1';

elsif (p_state = draw_state) then
      draw <= '1';

end if;
end process state_machine;
end behavior;
```

```
-- vhdl model for the move_error_detect

--*************************************************************************
-- IO Interface Declaration
--*************************************************************************

entity move_error_detect is
port
(
-- inputs
  signal   f:       in  vlbit_1d(8 downto 0);
  signal   p_move:  in  vlbit_1d(8 downto 0);

-- outputs
  signal  move_error_flag:  out vlbit
);
end move_error_detect;

--*************************************************************************
-- Architecture body
--*************************************************************************

architecture behavior of move_error_detect is

signal temp : vlbit_1d(8 downto 0);

begin
-- note no process is needed
-- note the temp signal to simplify my expression better than writing
-- the following
-- move_error <= (p_move(0) and f(0)) or (p_move(1) and f(1)) or ....

temp <= p_move and f;
move_error_flag <= temp(0) or temp(1) or temp(2) or temp(3) or temp(4)
                   or temp(5) or temp(6) or temp(7) or temp(8);

end behavior;
```

```
-- vhdl model for the mux2to1x9

--*************************************************************************
-- IO Interface Declaration
--*************************************************************************

entity mux2to1x9 is
port
(
-- inputs
  signal    a:  in  vlbit_1d(8 downto 0);
  signal    b:  in  vlbit_1d(8 downto 0);
  signal  sel:  in  vlbit;

-- outputs
  signal dout:  out vlbit_1d(8 downto 0)
);
end mux2to1x9;

--*************************************************************************
-- Architecture body
--*************************************************************************

architecture behavior of mux2to1x9 is
begin
  process (a, b, sel)
  begin

    if (sel = '1') then
       dout <= a;

    elsif (sel = '0') then
       dout <= b;

    end if;

  end process;

end behavior;
```

```
-- vhdl model for the other

--*************************************************************************
-- IO Interface Declaration
--*************************************************************************

entity other is
port
(
-- inputs
  signal   f: in vlbit_1d(8 downto 0);
  signal   p: in vlbit_1d(8 downto 0);

-- outputs
  signal c_move:     out vlbit_1d(8 downto 0)
);
end other;

--*************************************************************************
-- Architecture body
--*************************************************************************

architecture behv of other is
begin
process (p, f)
begin

-- include the defaults here or the outputs will be latched to previous out
   c_move(0) <= '0';
   c_move(1) <= '0';
   c_move(2) <= '0';
   c_move(3) <= '0';
   c_move(4) <= '0';
   c_move(5) <= '0';
   c_move(6) <= '0';
   c_move(7) <= '0';
   c_move(8) <= '0';



-- prevent possible loosing combinations here

-- case 1 possible losses
if    ( f(1) = '0' and
        (
            (p(0) = '1' and p(8) = '1')
         or (p(5) = '1' and p(6) = '1')
        )
      ) then
      c_move(1) <= '1';

-- case 2 & 3 possible losses
elsif ( f(8) = '0' and
        (
            (p(5) = '1' and p(7) = '1')
         or (p(5) = '1' and p(7) = '1')
         or (p(0) = '1' and p(7) = '1')
         or (p(5) = '1' and p(6) = '1')
        )
      ) then
      c_move(8) <= '1';

-- now take moves in order of
-- center
elsif (f(4) = '0') then c_move(4) <= '1';
```

```
-- corners
elsif (f(0) = '0') then c_move(0) <= '1';
elsif (f(2) = '0') then c_move(2) <= '1';
elsif (f(6) = '0') then c_move(6) <= '1';
elsif (f(8) = '0') then c_move(8) <= '1';

-- edges
elsif (f(1) = '0') then c_move(1) <= '1';
elsif (f(3) = '0') then c_move(3) <= '1';
elsif (f(5) = '0') then c_move(5) <= '1';
elsif (f(7) = '0') then c_move(7) <= '1';


-- note the following lines are not needed because we set the defaults
-- in the beginning if they are put in then you have wasted silicon
-- so be careful how and what your code does

--else
--   c_move(0) <= '0';
--   c_move(1) <= '0';
--   c_move(2) <= '0';
--   c_move(3) <= '0';
--   c_move(4) <= '0';
--   c_move(5) <= '0';
--   c_move(6) <= '0';
--   c_move(7) <= '0';
--   c_move(8) <= '0';


end if;

end process;
end behv;
```

```
-- vhdl model for the p_ready_gen

--*************************************************************************
-- IO Interface Declaration
--*************************************************************************

entity p_ready_gen is
port
(
-- inputs
  signal   p_move:  in  vlbit_1d(8 downto 0);

-- outputs
  signal  p_ready:  out vlbit
);
end p_ready_gen;

--*************************************************************************
-- Architecture body
--*************************************************************************

architecture behavior of p_ready_gen is
begin
 -- note no process


p_ready <= p_move(0) or p_move(1) or p_move(2) or p_move(3) or p_move(4)
           or p_move(5) or p_move(6) or p_move(7) or p_move(8) ;

end behavior;
```

```
-- vhdl model for the rdaw_detect

--*************************************************************************
-- IO Interface Declaration
--*************************************************************************

entity rdaw_detect is
port
(
-- inputs
  signal   f:  in  vlbit_1d(8 downto 0);

-- outputs
  signal  rdaw_flag:  out vlbit
);
end rdaw_detect;

--*************************************************************************
-- Architecture body
--*************************************************************************

architecture behavior of rdaw_detect is
begin
-- note no process is needed

rdaw_flag <= f(0) and f(1) and f(2) and f(3) and f(4)
             and f(5) and f(6) and f(7) and f(8) ;

end behavior;
```

```
library synth, work;
use synth.stdsynth.all, work.techdef.all;
entity tttmodule is
port(signal p_move:             in vlbit_1d(8 downto 0);
        signal c_move_1st:      in vlbit;
        signal clk:             in vlbit;
        signal reset:           in vlbit;
        signal x:               in vlbit;
        signal p:               inout vlbit_1d(8 downto 0);
        signal c:               inout vlbit_1d(8 downto 0);
        signal move_error:      out vlbit;
        signal c_win:           out vlbit;
        signal draw:    out vlbit);

end tttmodule;

architecture STRUCTURAL of tttmodule is

signal  v1:     vlbit_1d(8 downto 0);
signal  v2:     vlbit;
signal  v3:     vlbit;
signal  v5:     vlbit_1d(8 downto 0);
signal  v6:     vlbit_1d(8 downto 0);
signal  v7:     vlbit_1d(8 downto 0);
signal  v8:     vlbit;
signal  v9:     vlbit;
signal  v10:    vlbit;
signal  v11:    vlbit;
signal  v12:    vlbit;
signal  v13:    vlbit;
signal  v14:    vlbit_1d(8 downto 0);
signal  v15:    vlbit_1d(8 downto 0);
signal  v16:    vlbit_1d(8 downto 0);
signal  v17:    vlbit_1d(8 downto 0);


component board
port(
  signal    clk:          in  vlbit;
  signal    c_move:       in  vlbit_1d(8 downto 0);
  signal    load_c_move:  in  vlbit;
  signal    load_p_move:  in  vlbit;
  signal    p_move:       in  vlbit_1d(8 downto 0);
  signal    reset:        in  vlbit;
  signal    c:           inout  vlbit_1d(8 downto 0);
  signal    f:            out  vlbit_1d(8 downto 0);
  signal    p:           inout  vlbit_1d(8 downto 0));
end component;

component control
port(
-- inputs
  signal  clk:              in  vlbit;
  signal  c_move_1st:       in  vlbit;
  signal  draw_flag:        in  vlbit;
  signal  have_move_flag:   in  vlbit;
  signal  move_error_flag:  in  vlbit;
  signal  p_ready_flag:     in  vlbit;
  signal  reset:            in  vlbit;

-- outputs
  signal  c_move_control:  out  vlbit;
  signal  c_p_control:     out  vlbit;
  signal  c_win:           out  vlbit;
  signal  draw:            out  vlbit;
  signal  load_c_move:     out  vlbit;
```

```
-- for Actel we need:
--   FALLING = 0, LOWTRUE = 1
-- for ITD standard cell we need:
--   FALLING = 1, LOWTRUE = 1
-- for Xilinx we need:
--   FALLING = 0, LOWTRUE = 0

PACKAGE techdef IS

   constant FALLING : vlbit := '0';
   constant LOWTRUE: vlbit := '0';

END techdef;

PACKAGE BODY techdef IS

END techdef;
```

```
  signal  load_p_move:    out  vlbit;
  signal  move_error:     out  vlbit
);
end component;

component move_error_detect
port(
-- inputs
  signal   f:       in  vlbit_1d(8 downto 0);
  signal   p_move:  in  vlbit_1d(8 downto 0);

-- outputs
  signal  move_error_flag:  out vlbit
);
end component;

component mux2to1x9
port(
-- inputs
  signal    a:  in  vlbit_1d(8 downto 0);
  signal    b:  in  vlbit_1d(8 downto 0);
  signal  sel:  in  vlbit;

-- outputs
  signal dout:  out vlbit_1d(8 downto 0)
);
end component;

component other_move
port (
-- inputs
  signal   f: in vlbit_1d(8 downto 0);
  signal   p: in vlbit_1d(8 downto 0);

-- outputs
  signal c_move:     out vlbit_1d(8 downto 0)
);
end component;

component p_ready_gen
port(
-- inputs
  signal   p_move:  in  vlbit_1d(8 downto 0);

-- outputs
  signal  p_ready:  out vlbit
);
end component;

component rdaw_detect
port(
-- inputs
  signal   f:  in  vlbit_1d(8 downto 0);

-- outputs
  signal  rdaw_flag:  out vlbit
);
end component;

component win
port(
-- inputs
  signal c_p: in vlbit_1d(8 downto 0);
  signal   f: in vlbit_1d(8 downto 0);

-- outputs
```

```
  signal c_move:     out vlbit_1d(8 downto 0);
  signal x:             in vlbit;
  signal have_move:  out vlbit
);
end component;

begin

U1 : board
        port map (clk, v1, v2, v3, p_move, reset, c, v6, p);

U2 : control

        port map (clk, c_move_1st, v8, v9, v10, v11, reset, v12, v13, c_win, draw, v2, v3, mo

U3 : move_error_detect

        port map (v14, p_move, v10);

U4 : mux2to1x9

        port map (c, p, v13, v17);

U5 : mux2to1x9

        port map (v15, v16, v12, v1);

U6 : other_move

        port map (v6, p, v16);

U7 : p_ready_gen

        port map (p_move, v11);

U8 : rdaw_detect

        port map (v6, v8);

U9 : win

        port map (v17, v6, v16, x, v9);

end STRUCTURAL;
```

```
LIBRARY synth;
USE synth.stdsynth.all;



entity winblock is
port
(
        signal c_p: in vlbit_1d(8 downto 0);
        signal   f: in vlbit_1d(8 downto 0);

        signal c_move: out vlbit_1d(8 downto 0);
        signal have_move:  out vlbit
);
end winblock;

--*************************************************************************
-- Architecture body
--*************************************************************************

architecture behv of winblock is
begin
process (c_p, f)
begin

-- defaults
have_move <= '1'; -- default is i have a move but if i fall out
                  -- of the if statements then i dont have a move
                  -- this reduces the logic so i dont include the
                  -- have_move <= '1' in each of the if
                  -- statements since they are sequential

c_move(0) <= '0';
c_move(1) <= '0';
c_move(2) <= '0';
c_move(3) <= '0';
c_move(4) <= '0';
c_move(5) <= '0';
c_move(6) <= '0';
c_move(7) <= '0';
c_move(8) <= '0';


-- sq0
if    ( f(0) = '0' and                      --empty position
         ( (c_p(1) = '1' and c_p(2) = '1') --possible row win/block
        or (c_p(3) = '1' and c_p(6) = '1') --possible col win/block
        or (c_p(4) = '1' and c_p(8) = '1') --possible diag win/block
         )
      ) then
    c_move(0) <= '1';

-- sq1
elsif ( f(1) = '0' and                      --empty position
         ( (c_p(0) = '1' and c_p(2) = '1') --possible row win/block
        or (c_p(4) = '1' and c_p(7) = '1') --possible col win/block
         )
      ) then
    c_move(1) <= '1';

-- sq2
elsif ( f(2) = '0' and                      --empty position
         ( (c_p(0) = '1' and c_p(1) = '1') --possible row win/block
        or (c_p(5) = '1' and c_p(8) = '1') --possible col win/block
        or (c_p(4) = '1' and c_p(6) = '1') --possible diag win/block
         )
```

```
      ) then
    c_move(2) <= '1';

-- sq3
elsif ( f(3) = '0' and                      --empty position
         ( (c_p(0) = '1' and c_p(6) = '1') --possible row win/block
        or (c_p(4) = '1' and c_p(5) = '1') --possible col win/block
         )
      ) then
    c_move(3) <= '1';

-- sq4
elsif ( f(4) = '0' and                      --empty position
         ( (c_p(1) = '1' and c_p(7) = '1') --possible row win/block
        or (c_p(3) = '1' and c_p(5) = '1') --possible col win/block
         )
      ) then
    c_move(4) <= '1';

-- sq5
elsif ( f(5) = '0' and                      --empty position
         ( (c_p(2) = '1' and c_p(8) = '1') --possible row win/block
        or (c_p(3) = '1' and c_p(4) = '1') --possible col win/block
         )
      ) then
    c_move(5) <= '1';

-- sq6
elsif ( f(6) = '0' and                      --empty position
         ( (c_p(7) = '1' and c_p(8) = '1') --possible row win/block
        or (c_p(0) = '1' and c_p(3) = '1') --possible col win/block
        or (c_p(2) = '1' and c_p(4) = '1') --possible diag win/block

         )
      ) then
    c_move(6) <= '1';

-- sq7
elsif ( f(7) = '0' and                      --empty position
         ( (c_p(6) = '1' and c_p(8) = '1') --possible row win/block
        or (c_p(1) = '1' and c_p(4) = '1') --possible col win/block
         )
      ) then
    c_move(7) <= '1';

-- sq8
elsif ( f(8) = '0' and                      --empty position
         ( (c_p(6) = '1' and c_p(7) = '1') --possible row win/block
        or (c_p(2) = '1' and c_p(5) = '1') --possible col win/block
        or (c_p(0) = '1' and c_p(4) = '1') --possible diag win/block
         )
      ) then
    c_move(8) <= '1';


else
have_move <= '0';

end if;

end process;
end behv;
```